Query Optimization

The query optimizer is the most important block in DBMS: it generates the strategy called execution plan. This module also defines how reads are performed:

. directly;

. with indices.

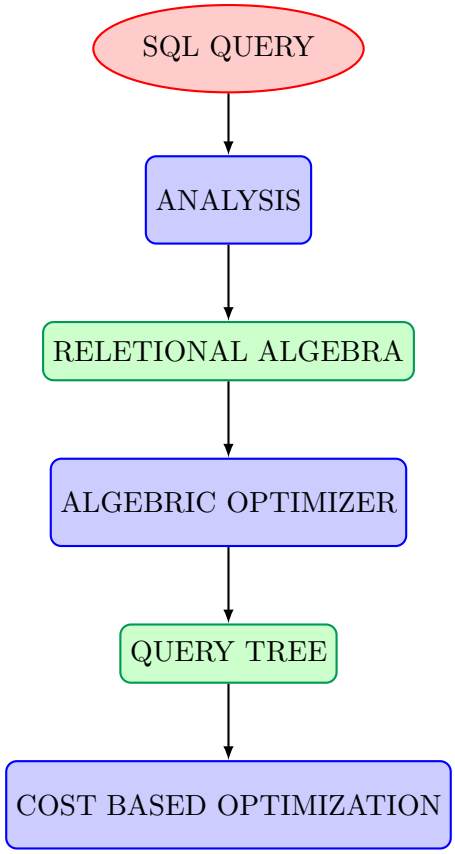
In order to minimize the execution cost is possible:

. minimize the number of reads;

. minimize the size of intermediates results.

All possibles plans are evaluated statistically: to have better performances strategies are periodically update.

Modules



Execution modes

If query is executed and the access program is immediatly generated is used *compile* & *go* mode; it is very useful when database and user interact frequently.

When the access program is not immedialty generated, many alternatives can be evaluated in order to provide the best result: this is *compile* & *store* mode.

Analysis

Analysis block exploits:

. lexical control;

. syntactic control;

. semantic control

in order to translate query received in relational algebra.

Algebraic Optimization

This module generates query tree from relational algebra indipendently by data but, probably the cost is different.

Most important properties are:

. intermediates results are reduced before being stored in memory;

. an expression for the application who reduced intermediates results is prepared.

Transformations with examples In order to exploits examples let consider the two tables:

. **employ**: (emp_n, ename, dept_n, salary);

. **department**: (dept_n, dname).

Atomization of selection In general:

$$\sigma_{F_1 \wedge F_2}(T) \equiv \sigma_{F_1}(\sigma_{F_2}(T)) \equiv \sigma_{F_2}(\sigma_{F_1}(T))$$

In the example:

$$\sigma_{name='rossi' \wedge salary>1000}(Em) \equiv \sigma_{name}(\sigma_{salary}(Em)) \equiv \sigma_{name}(\sigma_{salary}(Em))$$

Cascading projections In general:

$$\pi_x(T) = \pi_x(\pi_{x,y}(T))$$

In the example:

$$\pi_{emp\_n}(Em) = \pi_{emp\_n}(\pi_{emp\_n,\, ename}(Em))$$

Anticipation of selection with respect to join In general:

$$\sigma_F(T_1 \bowtie T_2) = T_1 \bowtie (\sigma_F(T_2))$$

In the example, consider the join of:

$$Em \bowtie_p Dep$$

where:

$$p : emp.dept\_n = dept.dept\_n$$

This operation:

$$\sigma_{dname=delen}(Em \bowtie_p Dep) = Em \bowtie_p (\sigma_{dname=delen}(Dep))$$

Anticipation of projection with respect to join In general:

$$\pi_x(T_1 \bowtie T_2) = \pi_x((\pi_{x,y}(T_1) \bowtie (\pi_{x,y}(T_2)))$$

In the example:

$$\pi_{dname,ename}(Em \bowtie_p Dep) = \pi_{dname,ename}((\pi_{dept\_n,ename}(Em) \bowtie_p \\ \bowtie_p (\pi_{dept\_n,dname}(Dep)))$$

Join derivation from Cartesian product In general:

$$\sigma_F(T_1 \times T_2) = T_1 \bowtie_F T_2$$

Distribution of selection with respect to union In general:

$$\sigma_F(T_1 \cup T_2) = (\sigma_F(T_1)) \cup (\sigma_F(T_2))$$

Notice that, in order to have a positive result, two tables must be omogenous (same schema).

Distribution of selection with respect to difference In general:

$$\sigma_F(T_1 - T_2) = (\sigma_F(T_1)) - (\sigma_F(T_2))$$

Distribution of projection with respect to union In general:

$$\pi_x(T_1 \cup T_2) = (\pi_x(T_1)) \cup (\pi_x(T_2))$$

Distribution of projection with respect to difference In general:

$$\pi_x(T_1 - T_2) = (\pi_x(T_1)) - (\pi_x(T_2))$$

As selection, projection can have a distribution with respect to the difference too, only if x includes the primary key or a set of attributes with same properties of the primary key such as:

. being unique;

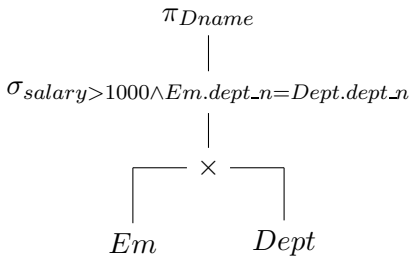
. being not null.

Example

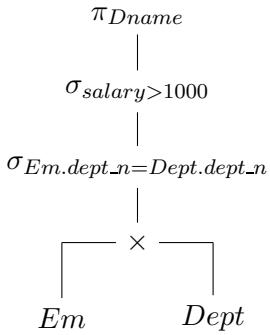
Using same tables described previously consider the following SQL query:

```
1 SELECT DISTINCT Dname
2 FROM Em, Dept
3 WHERE Em.dept_n=Dept.dept_n
4 AND Salary > 1000;
```

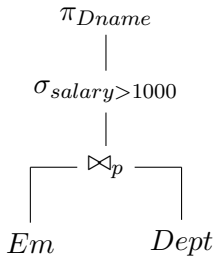
The canonical form is:



The first transformation is done separating the two conditions of selection:



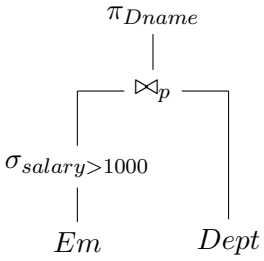
The second transformation change the cartesian product in a join operation:



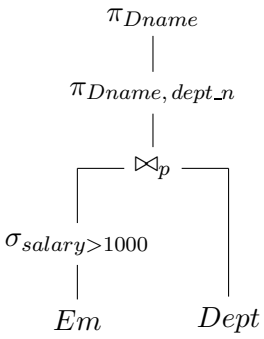
where:

$$p : emp.dept\_n = dept.dept\_n$$

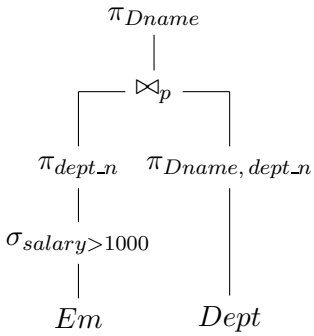
The third transformation is anticipate selection:



Last two tranformations are done in order to anticipate projection; first the projection is decomposed:



then projection is anticipate:



Cost Based Optimization

This module has to choose between all *canonical query trees* processed by the Algebraic Optimizer. It is based on two parts:

. data profiles which are statistical information describing tables and expressions, for example number and size (in bytes) of tuples, number and size (in bytes too) of attributes; this information, located in data dictionary, is periodically updated and is called *table profiles*;

. cost of access operations.

Access Operators

The function provided is select an option in the leaf node of query trees that optimizer has built.

Full scan This approach is a sequential scan of the table: is very useful when there are no indices or when indices does not reduce the reading cost. Otherwise there are operations who can reduce the size of data to read.

Predicate evaluation Access to table is realized through indices: the predicate must have a good *selectivity* expecially for unclustered indices otherwise the memory block does not fit in main memory. Is possible that, if a predicate has bad selectivity, the best operation is a full scan read.

Conjunction of predicates The condition is:

$$A_i = v_1 \wedge A_j = v_2$$

The predicate who has the best selectivity is first applied and then the second one is applied. It is also possible to build a composed index using both predicates: pointers are sorted on more fields, but complexity increases a lot.

Disjunction of predicates The condition is:

$$A_i = v_1 \vee A_j = v_2$$

Indices, if all predicates are supported by an index, can be used otherwise is better a full scan because both attributes compare.

Join operation This operation is very critical because connect two tables is based on values and, typically all results (intermediates and final) are larger than initial tables. There are three different algorithms:

. nested loop;

. merge scan join;

. hash join.

Nested loop This algorithm distinguish tables in:

. inner table;

. outer table.

Join operation is realized reading once the outer table and, for each tuple, read the entire inner table in order to find matches. This approach is also called *brute force*.

The efficiency is due to the size of the inner table: if it is small or indexed the cost is not so expensive. For these reasons the inner table is chosen as the smaller of the two: the technique is not symmetric because there will be different cost changing options.

Merge scan With this algorithm the selection of inner and outer table does not have effecto to the cost of execution: first of all both tables are sorted on the join attributes, then they are read in parallel and tuples present in both tables are generated on correspondig values.

Matching tuples are located in a few countinous memory blocks and since there is only one reading this approach is faster than nested loop. The disadvantage is sorting required on both tables, but if the physical structure is one already sorted on join attributes the drawback is not relevant.

Hash Join This is the fastest technique because perform hash function is fast than sort or read operations. The algorithm apply the same hash function to the join attribute in both tables so tuples witch have the same value will be put in the same memory block. It is important notice that in each bucket tuples must be ordered so a local sort is required.

Group by The group by close can be performed in two ways:

. sorted based by attributes, then aggregate on groups;

. hash based when an hash function is applied on the group of attributes.

Execution plan selection To find the optimal execution plan the following dimensions are:

. how data is read;

. the execution order of all operations;

. the technique in witch each operator is implemented.

. how often sorts are performed.

It is possible that if a join operation is performed with a merge scan the best choice is having the group by implemented with sort; another example is join done with hash approach, so group by will be done with hash method. The previous operation can support others later operation.

The optimizer builds alternatives in a tree, where:

. intermediate nodes makes decision on a variable;

. leaf nodes are the final query execution plan.